CS143A
Principles on Operating Systems
Discussion 09:

Instructor: Prof. Anton Burtsev

TA: Saehanseul Yi (Hans)

Dec 6, 2019 **1pm**

Agenda

- Implementing new system calls
- Threads

Files to modify for new system calls

- syscall.h – SYS_wrprotect
- syscall.c – sys_wrprotect
- user.h – wrprotect
- usys.S – SYSCALL(wrprotect)

```
22 int read(int, void*, int);
21 int close(int);
20 int kill(int);
19 int exec(char*, char**);
18 int open(const char*, int);
17 int wrprotect(void* addr, int);
16 int mknod(const char*, short, short);
15 int unlink(const char*);
```

<user.h>

label

similar to 'goto' syntax

```
1  #define SYSCALL(name) \
2    .globl name; \
3    name: \
4      movl $SYS_ ## name, %eax; \
5      int $T_SYSCALL; \
6      ret
```

<usys.S>

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

<trap.c>

Threads

- Lightweight Process (LWP)
  - They share address space
  - We don’t create new pages—faster creation
- Inter-process communication is costly:
  - Through file
  - Shared memory (much complicated than thread’s)
  - pipe
  - socket
  - …
- Faster context-switching

Creating threads

```
181 fork(void)
182 {
183   int i, pid;
184   struct proc *np;
185   struct proc *curproc = myproc();
186
187   // Allocate process.
188   if((np = allocproc()) == 0){
189     return -1;
190   }
191
192   // Copy process state from proc.
193   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
194     kfree(np->kstack);
195     np->kstack = 0;
196     np->state = UNUSED;
197     return -1;
198   }
199   np->sz = curproc->sz;
200   np->parent = curproc;
201   *np->tf = *curproc->tf;
202
203   // Clear %eax so that fork returns 0 in the child.
204   np->tf->eax = 0;
```

create process

Creating threads

```
181 fork(void)
182 {
183   int i, pid;
184   struct proc *np;
185   struct proc *curproc = myproc();
186
187   // Allocate process.
188   if((np = allocproc()) == 0){
189     return -1;
190   }
191
192   // Copy process state from proc.
193   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
194     kfree(np->kstack);
195     np->kstack = 0;
196     np->state = UNUSED;
197     return -1;
198   }
199   np->sz = curproc->sz;
200   np->parent = curproc;
201   *np->tf = *curproc->tf;
202
203   // Clear %eax so that fork returns 0 in the child.
204   np->tf->eax = 0;
```

create process

allocate user virtual memory & copy pages
(we don't need this)

Creating threads

```
181 fork(void)
182 {
183   int i, pid;
184   struct proc *np;
185   struct proc *curproc = myproc();
186
187   // Allocate process.
188   if((np = allocproc()) == 0){
189     return -1;
190   }
191
192   // Copy process state from proc.
193   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
194     kfree(np->kstack);
195     np->kstack = 0;
196     np->state = UNUSED;
197     return -1;
198   }
199   np->sz = curproc->sz;
200   np->parent = curproc;
201   *np->tf = *curproc->tf;
202
203   // Clear %eax so that fork returns 0 in the child.
204   np->tf->eax = 0;
```

create process

allocate user virtual memory & copy pages
(we don't need this)

Creating threads

```
201   *np->tf = *curproc->tf;
202
203   // Clear %eax so that fork returns 0 in the child.
204   np->tf->eax = 0;
205
206   for(i = 0; i < NOFILE; i++)
207     if(curproc->ofile[i])
208       np->ofile[i] = filedup(curproc->ofile[i]);
209   np->cwd = idup(curproc->cwd);
210
211   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
212
213   pid = np->pid;
214
215   acquire(&ptable.lock);
216
217   np->state = RUNNABLE;
218
219   release(&ptable.lock);
220
221   return pid;
222 }
```

return value.. where does eip point to now?

Creating threads

```
201   *np->tf = *curproc->tf;
202
203   // Clear %eax so that fork returns 0 in the child.
204   np->tf->eax = 0;
205
206   for(i = 0; i < NOFILE; i++)
207     if(curproc->ofile[i])
208       np->ofile[i] = filedup(curproc->ofile[i]);
209   np->cwd = idup(curproc->cwd);
210
211   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
212
213   pid = np->pid;
214
215   acquire(&ptable.lock);
216
217   np->state = RUNNABLE;
218
219   release(&ptable.lock);
220
221   return pid;
222 }
```

return value.. where does eip point to now?

copy file descriptors

Creating threads

```
201   *np->tf = *curproc->tf;
202
203   // Clear %eax so that fork returns 0 in the child.
204   np->tf->eax = 0;
205
206   for(i = 0; i < NOFILE; i++)
207     if(curproc->ofile[i])
208       np->ofile[i] = filedup(curproc->ofile[i]);
209   np->cwd = idup(curproc->cwd);
210
211   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
212
213   pid = np->pid;
214
215   acquire(&ptable.lock);
216
217   np->state = RUNNABLE;
218
219   release(&ptable.lock);
220
221   return pid;
222 }
```

return value.. where does eip point to now?

copy file descriptors

schedule the thread

Creating threads

```
201   *np->tf = *curproc->tf;
202
203   // Clear %eax so that fork returns 0 in the child.
204   np->tf->eax = 0;
205
206   for(i = 0; i < NOFILE; i++)
207     if(curproc->ofile[i])
208       np->ofile[i] = filedup(curproc->ofile[i]);
209   np->cwd = idup(curproc->cwd);
210
211   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
212
213   pid = np->pid;
214
215   acquire(&ptable.lock);
216
217   np->state = RUNNABLE;
218
219   release(&ptable.lock);
220
221   return pid;
222 }
```

return value.. where does eip point to now?

copy file descriptors

kind of a spinlock

schedule the thread

Creating thread

```
201   *np->tf = *curproc->tf;
202
203   // Clear %eax so that fork returns 0 i
204   np->tf->eax = 0;
205
206   for(i = 0; i < NOFILE; i++)
207     if(curproc->ofile[i])
208       np->ofile[i] = filedup(curproc->of
209   np->cwd = idup(curproc->cwd);
210
211   safestrcpy(np->name, curproc->name, si
212
213   pid = np->pid;
214
215   acquire(&ptable.lock);
216
217   np->state = RUNNABLE;
218
219   release(&ptable.lock);
220
221   return pid;
222 }
```

```
struct trapframe {
  // registers as pushed b
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // usele
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when
  uint esp;
  ushort ss;
  ushort padding6;
```

return value.. where does eip point to now?

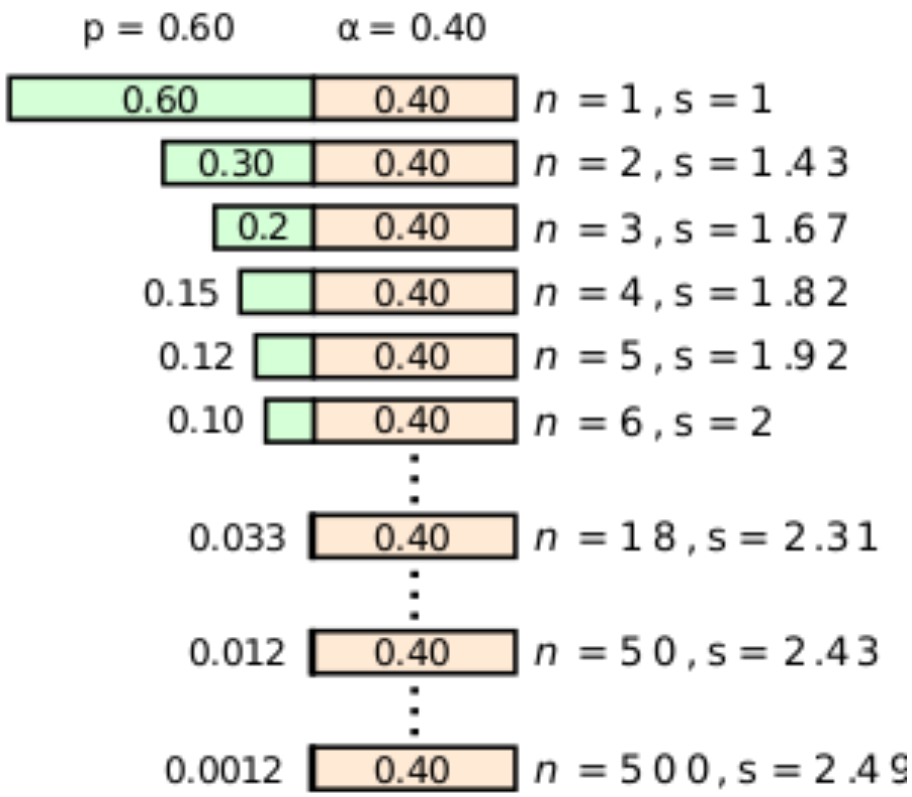
copy file descriptors

1. set eip
2. set esp (stack top)
3. push arguments and return address to stack

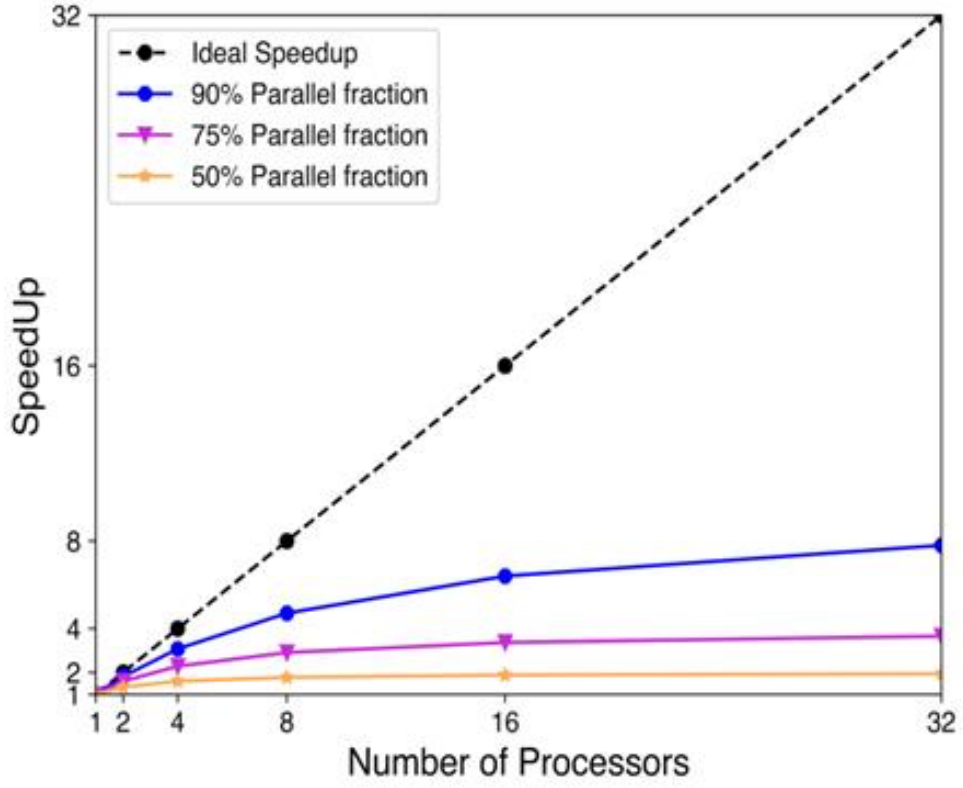
kind of a spinlock

schedule the thread

Amdahl's Law



- A program consists of two parts: parallelizable vs. non-parallelizable(serial)
- If parallelizable takes 60% of execution time, even if we use infinite number of threads to parallelize it, the program's execution time is reduced by 60%
- maximum speedup 1/0.4 = 2.5x



Race condition (Data race)

Parent
a=0;

Thread #1
b=a++;

Thread #2
b=a++;

Thread 1	Thread 2	Memory Value
read		0
increment		0
write		1
	read	1
	increment	1
	write	2

Thread#1's b: 1
Thread#2's b: 2

Race condition (Data race)

Parent
a=0;

Thread #1
b=a++;

Thread #2
b=a++;

Thread 1	Thread 2	Memory Value
read		0
increment		0
write		1
	read	1
	increment	1
	write	2

Thread 1	Thread 2	Memory Value
read		0
	read	0
	increment	0
	write	1
increment		1
write		1

Thread#1's b: 1
Thread#2's b: 2

Race condition (Data race)

Parent
a=0;

Thread #1
b=a++;

Thread #2
b=a++;

Thread 1	Thread 2	Memory Value
read		0
increment		0
write		1
	read	1
	increment	1
	write	2

Thread#1's b: 1
Thread#2's b: 2

Thread 1	Thread 2	Memory Value
read		0
	read	0
	increment	0
	write	1
increment		1
write		1

Thread#1's b: 1
Thread#2's b: 1

Race condition: Atomic operations

Parent
a=0;

Thread #1
b=a++;

Thread #2
b=a++;

Process 1	Process 2	Memory Value
atomic_inc		1
	atomic_inc	2

atomic_inc = read + inc + write

Thread#1's b:1
Thread#2's b: 2

Race condition: Atomic operations

Parent
a=0;

Thread #1
b=a++;

Thread #2
b=a++;

Process 1	Process 2	Memory Value
atomic_inc		1
	atomic_inc	2

atomic_inc = read + inc + write

- Special instructions
- e.g. xchg

temp = a;
a = b;
b = temp;

temp = a;
a = b;
b = temp;

Thread#1's b: 1
Thread#2's b: 2

Race condition: Atomic operations

Parent
a=0;

Thread #1
b=a++;

Thread #2
b=a++;

Process 1	Process 2	Memory Value
atomic_inc		1
	atomic_inc	2

atomic_inc = read + inc + write

Thread#1's b: 1
Thread#2's b: 2

- Special instructions
- e.g. xchg

temp = a;
a = b;
b = temp;

temp = a;
a = b;
b = temp;

Lock()
… (Critical section)
Unlock()

```
void Lock()
{
   while (lock ==1);
   lock = 1;
}

void Unlock()
{
    lock = 0;
}
```

Race condition: Atomic operations

Parent
a=0;

Thread #1
b=a++;

Thread #2
b=a++;

Thread 1	Thread 2	Memory Value
atomic_inc		1
	atomic_inc	2

atomic_inc = read + inc + write

Thread#1's b: 1
Thread#2's b: 2

- Special instructions
- e.g. xchg

```
temp = a;
a = b;
b = temp;
```

```
temp = a;
a = b;
b = temp;
```

```
Lock()
… (Critical section)
Unlock()
```

```
void Lock()
{
   while (lock ==1);
   lock = 1;
}

void Unlock()
{
    lock = 0;
}
```

Spinning!

Race condition: Atomic operations

- Spinlock uses CPU continuously
- It will take portion of CPU utilization
- Degrades performance of other threads/process

```
void Lock()
{
   while (lock ==1);
   lock = 1;
}

void Unlock()
{
    lock = 0;
}
```

Spinning!

Race condition: Atomic operations

- Spinlock uses CPU continuously
- It will take portion of CPU utilization
- Degrades performance of other threads/process
- **Why don’t we make it sleep while waiting?**

```
void Lock()
{
   while (lock ==1);
   lock = 1;
}

void Unlock()
{
    lock = 0;
}
```

Spinning!

Race condition: Atomic operations

- Spinlock uses CPU continuously
- It will take portion of CPU utilization
- Degrades performance of other threads/process
- **Why don’t we make it sleep while waiting?**

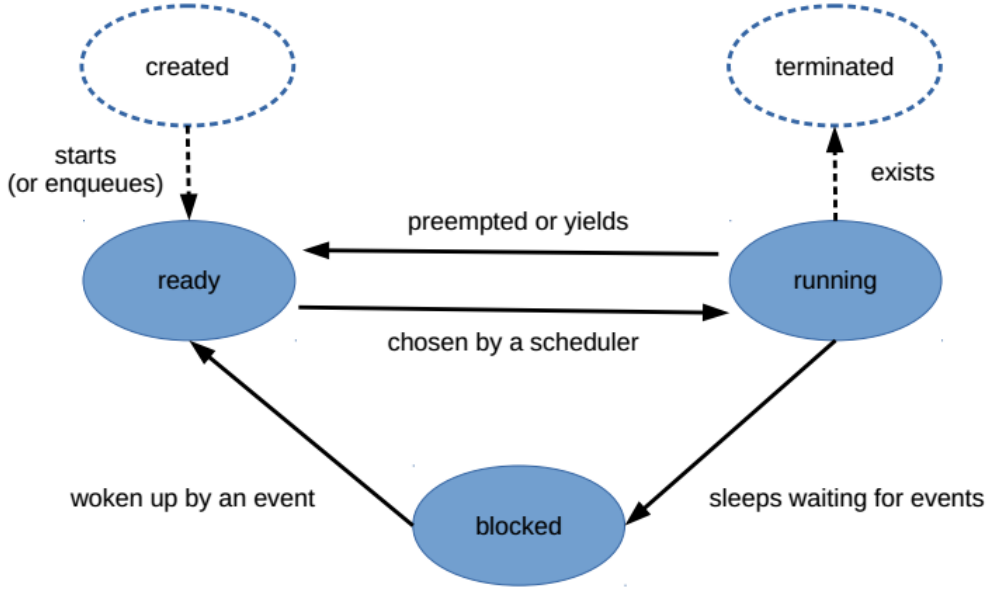
```
void Lock()
{
   while (lock ==1);
   lock = 1;
}

void Unlock()
{
    lock = 0;
}
```

Spinning!

Race condition: Atomic operations

- Spinlock uses CPU continuously
- It will take portion of CPU utilization
- Degrades performance of other threads/process
- **Why don't we make it sleep while waiting?**
  → Mutex (mutually exclusive)

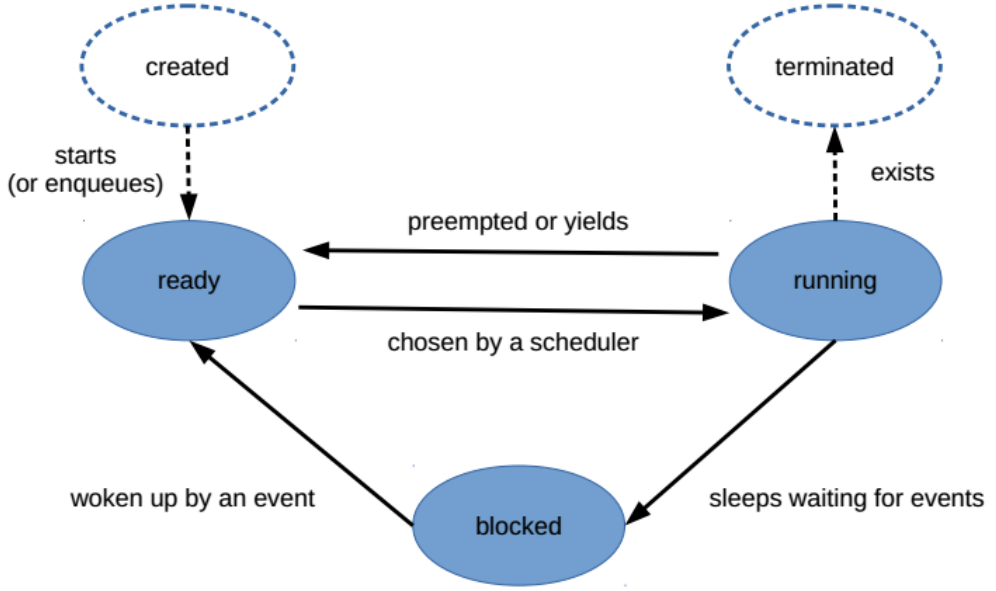
```
void Lock()
{
   while (lock ==1)
      yield();
   lock = 1;
}
```



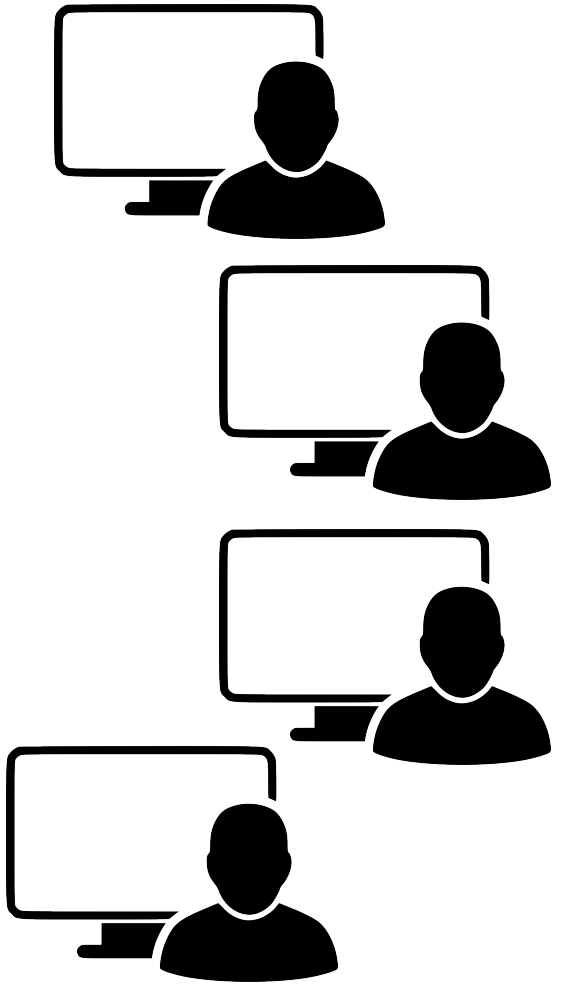
Race condition: Atomic operations

- Spinlock uses CPU continuously
- It will take portion of CPU utilization
- Degrades performance of other threads/process
- **Why don't we make it sleep while waiting?**
  → Mutex (mutually exclusive)
- **xv6 doesn't have yield() so we use sleep() to mimic the behavior**

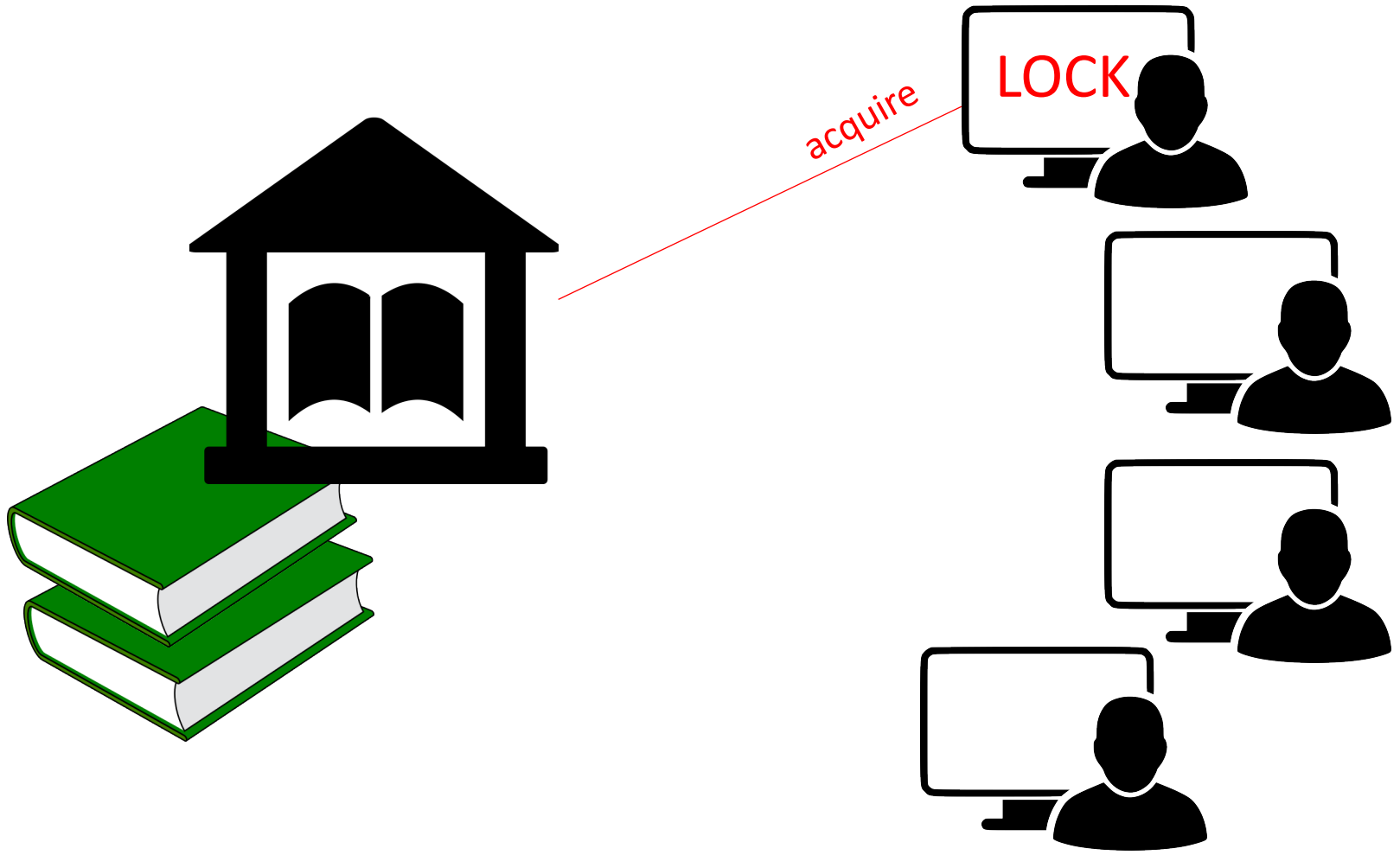
```
void Lock()
{
   while (lock ==1)
      sleep(1);
   lock = 1;
}
```



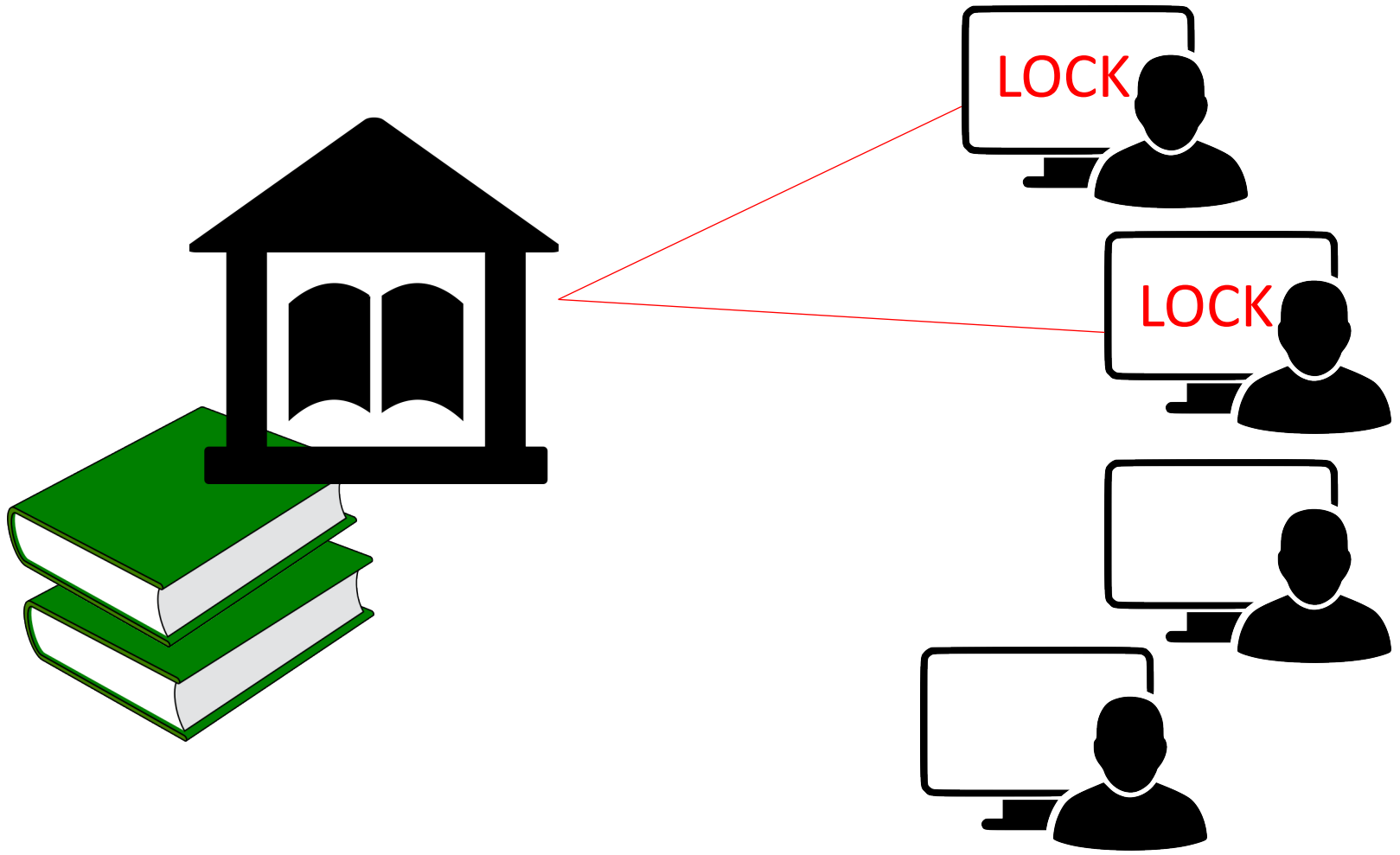
Semaphore: Producer--Consumer



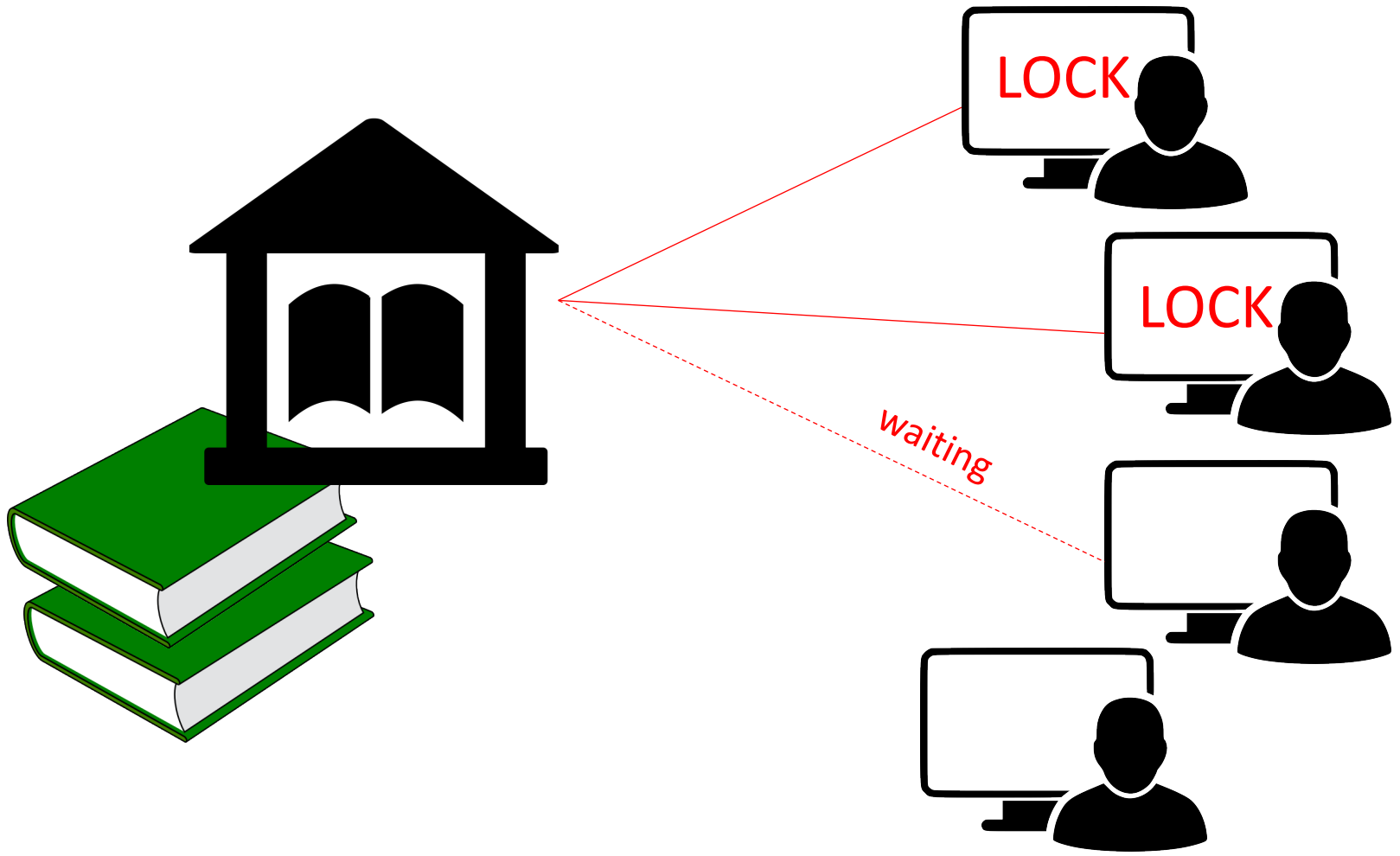
Semaphore: Producer--Consumer



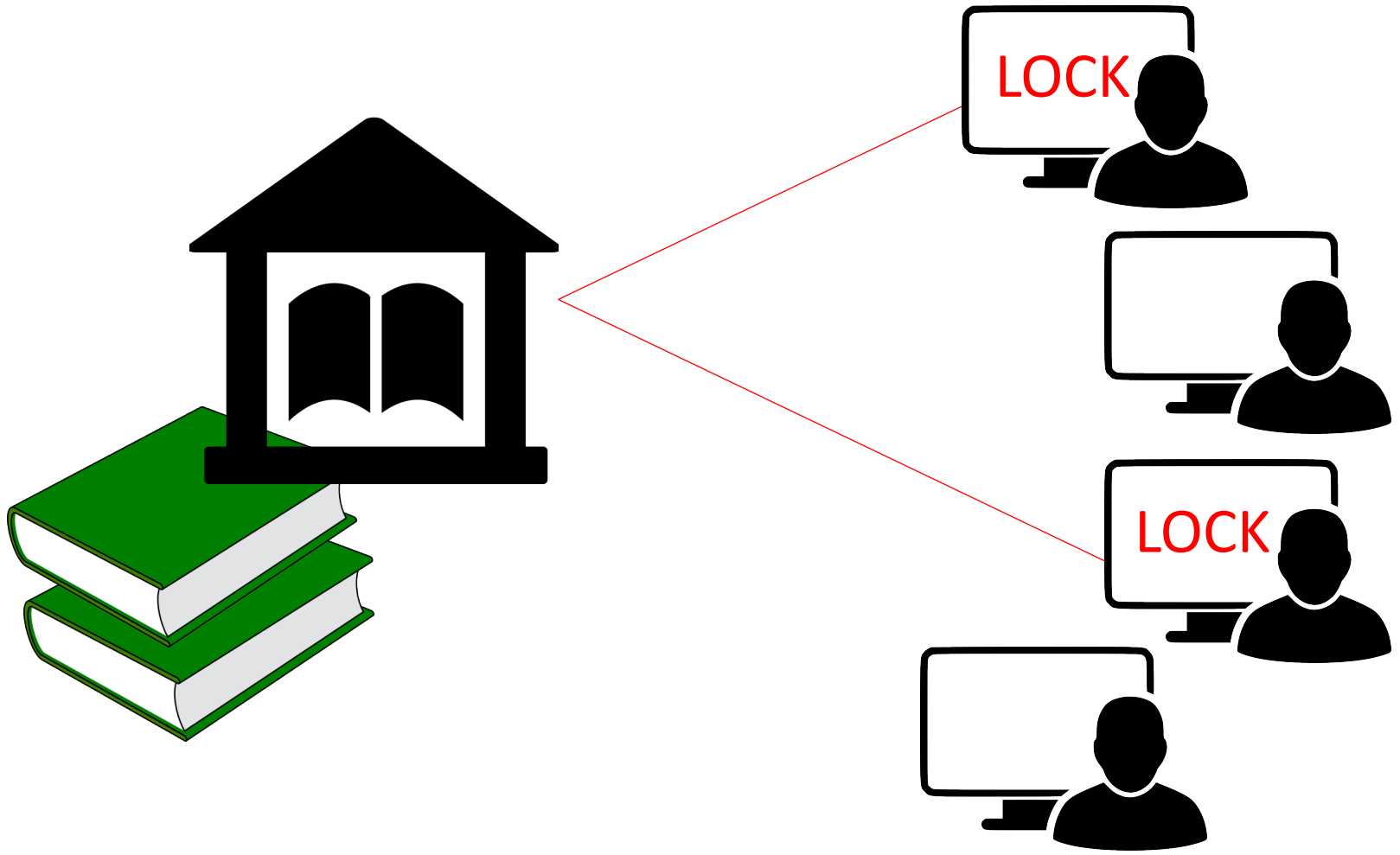
Semaphore: Producer--Consumer



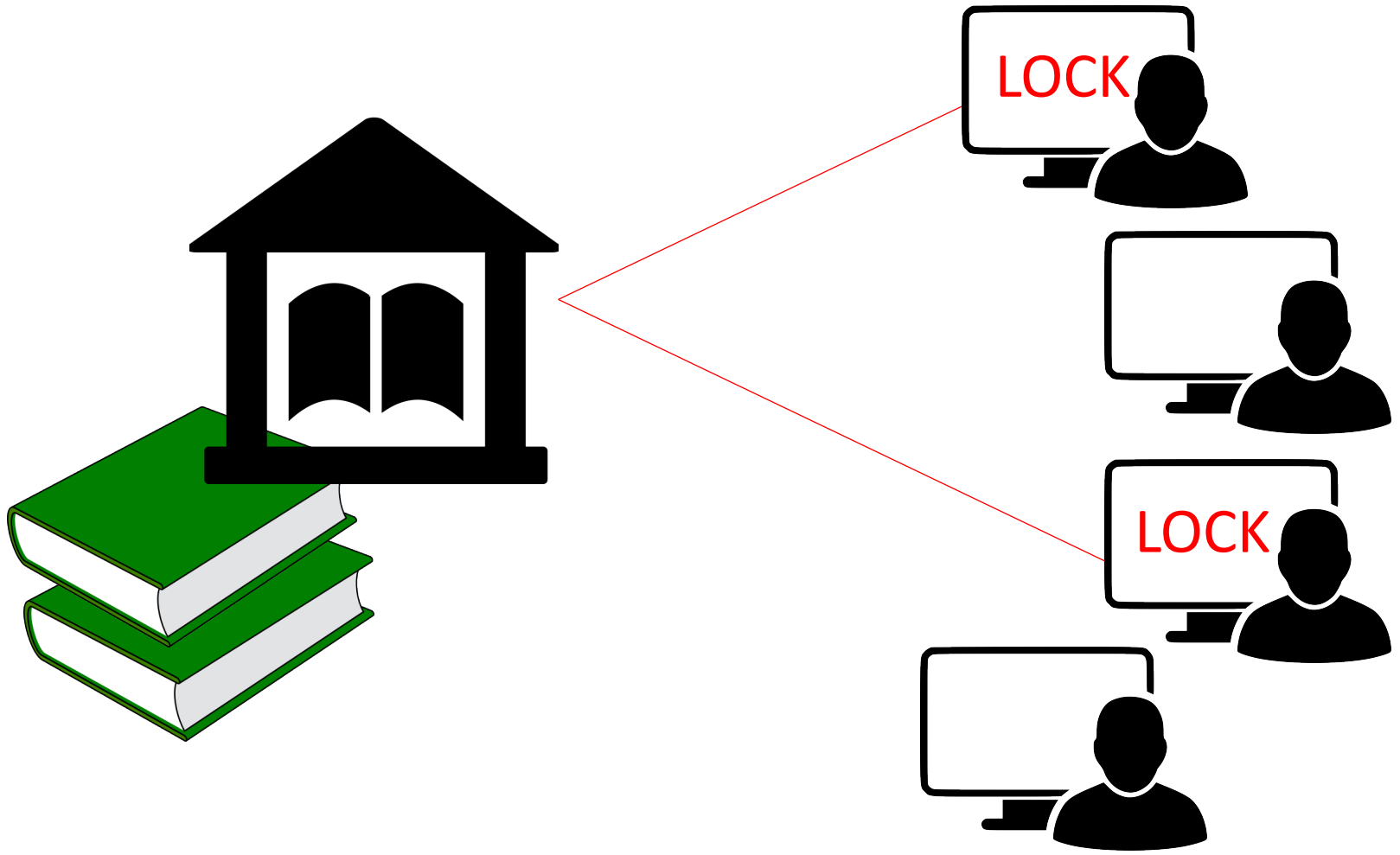
Semaphore: Producer--Consumer



Semaphore: Producer--Consumer



Semaphore: Producer--Consumer



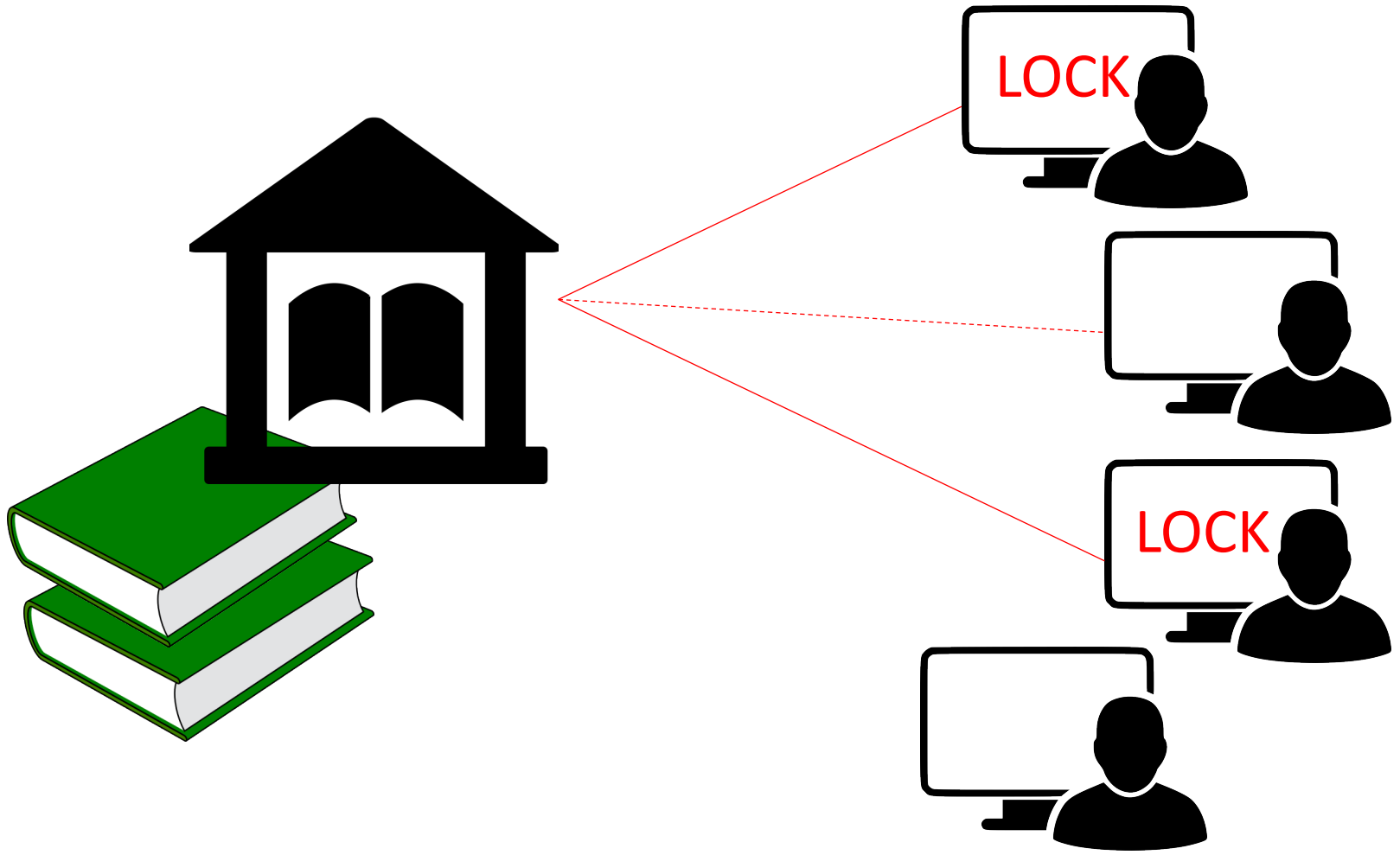
sem_init: initialize counter (~ # of books)

sem_wait: if counter > 0, decrease by 1
if counter == 0, wait until it is greater than 0

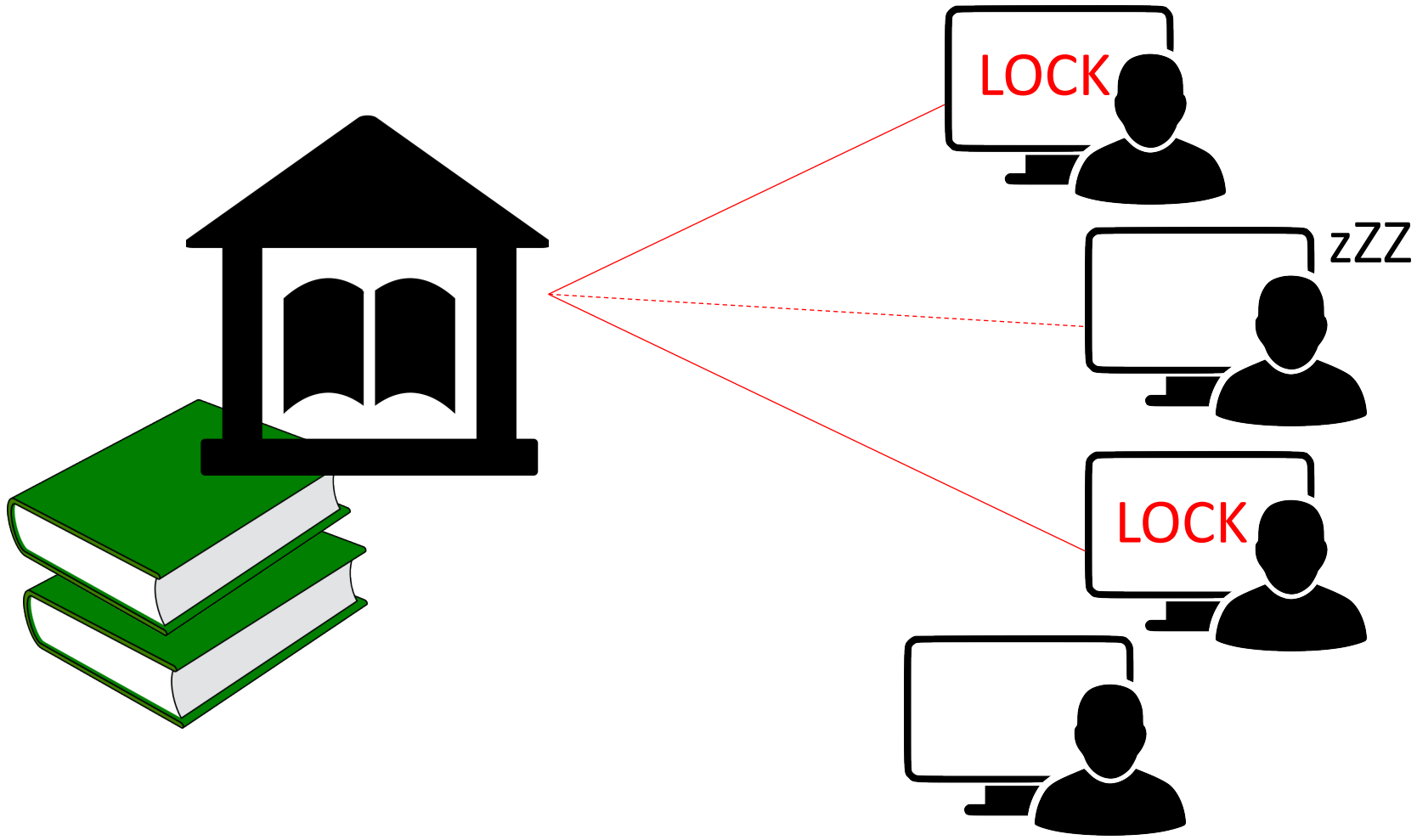
sem_post: increase counter by 1

HINT: counter should be in critical section.
You can also use cond var

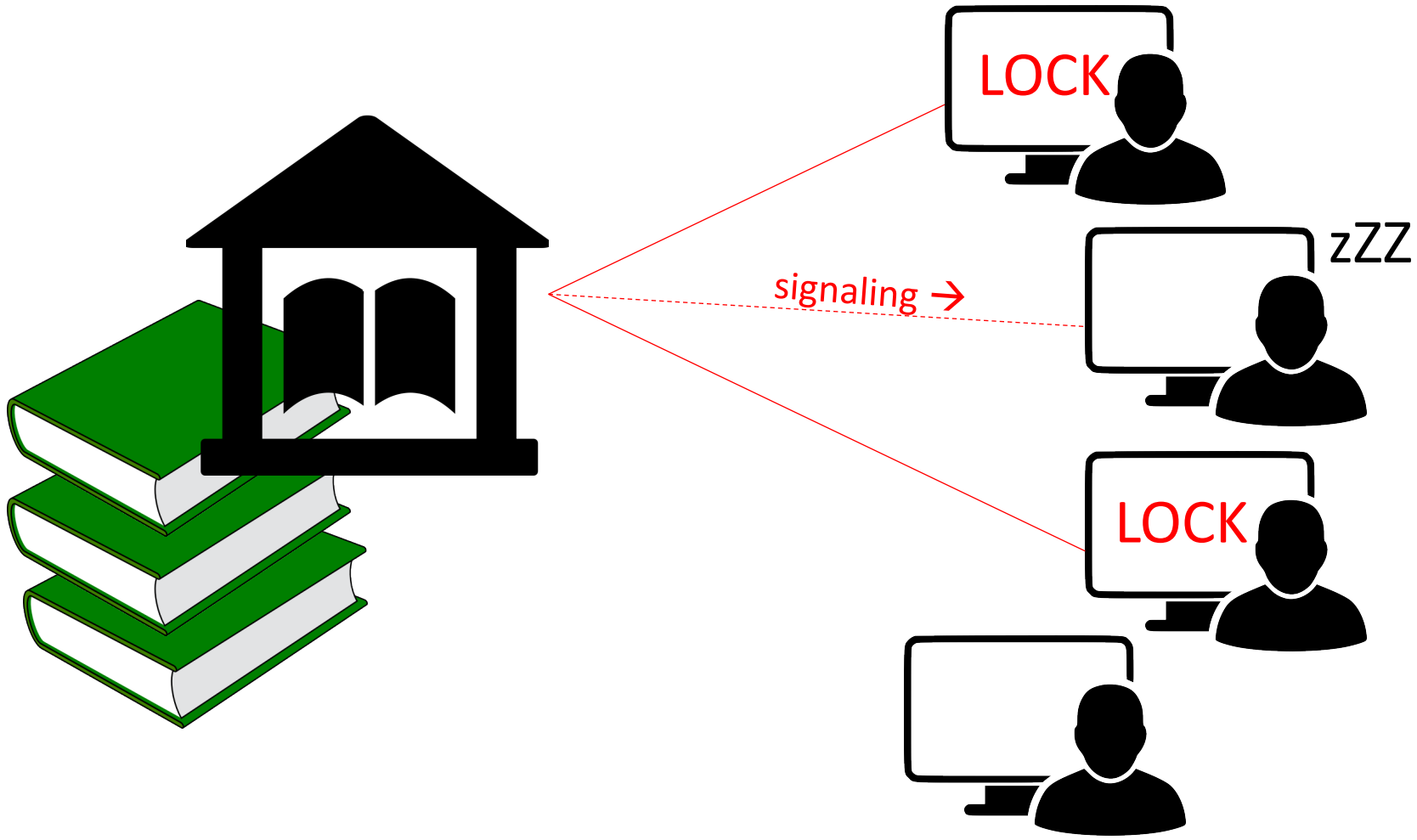
Conditional Variable: Producer--Consumer



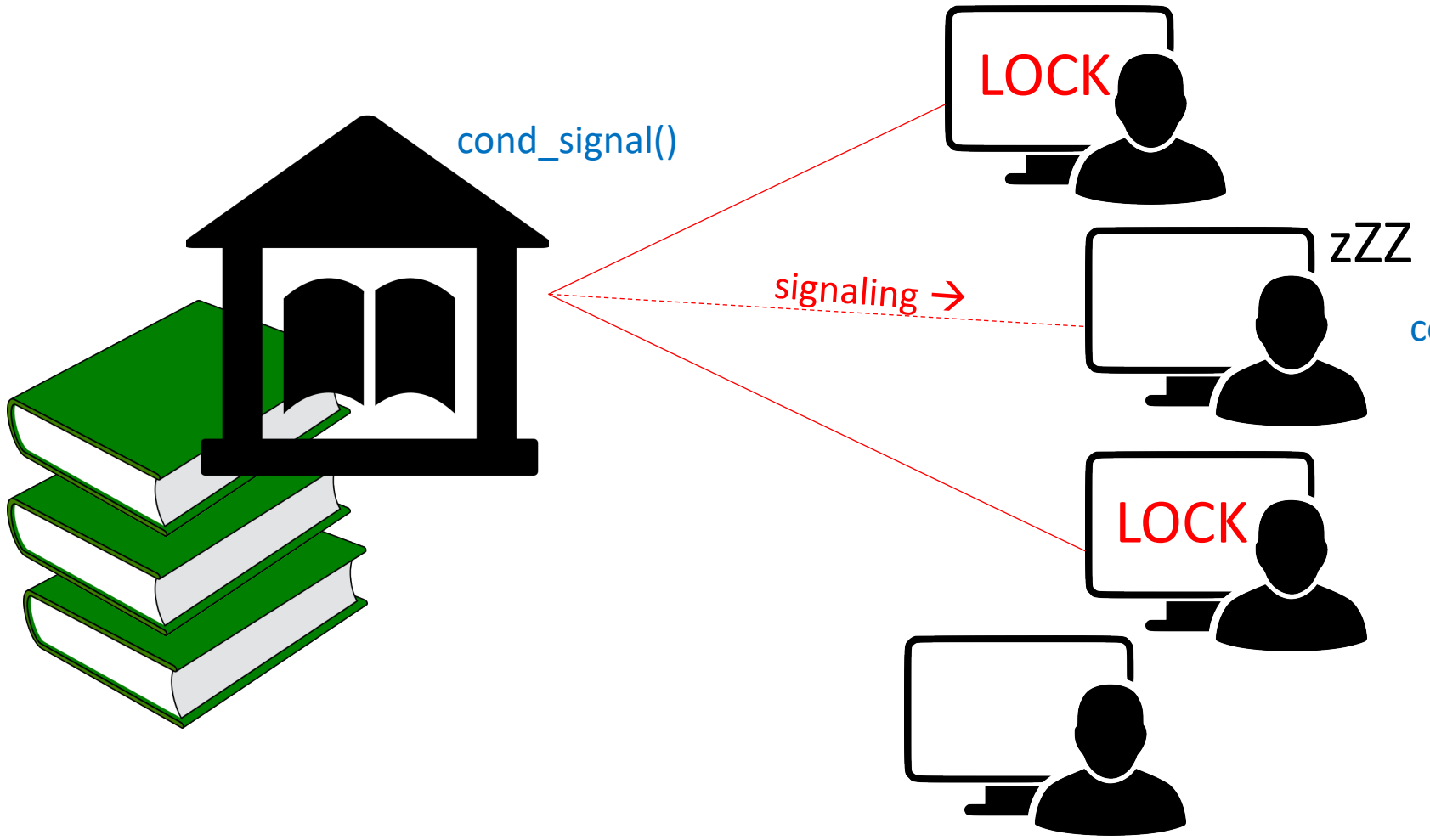
Conditional Variable: Producer--Consumer



Conditional Variable: Producer--Consumer



Conditional Variable: Producer--Consumer



- Create another variable ‘signal’
- User wakes up time to time and check if signal has come
- Otherwise go to sleep again

Per-thread Variables

- Local variables will be on stack..
- its address points to somewhere in the thread's stack