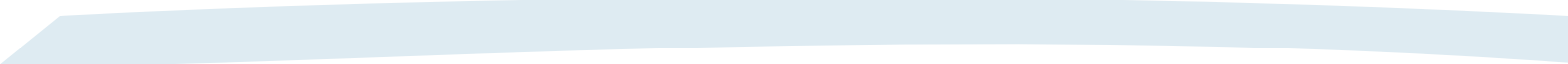


Introduction to Concurrency



CSE 120

Winter 2001

Keith Marzullo

Concurrency

A *process* is a program executing on a virtual computer.

Issues such as *processor speed* and *multiplexing of shared resources* are abstracted away.

... we assume that processes share memory (*thread* or *lightweight process*).

Creating processes (I)

A possible syntax:

```
ID fork(int (*proc)(int), int p);  
int join(ID i);
```

```
int foo(int p);  
int r;  
ID i = fork(foo, 3);  
...  
r = join(i);
```

Creating processes (II)

Another syntax:

cobegin

shared variable declaration
code for process 1

||

code for process 2

||

...

coend

Properties

Concurrent programs are specified using *properties*, which is a predicate that evaluated over a run of the concurrent program.

A property: the value of x is always at least as large as that of y and has the value of 0 at least once.

Not a property: the average number of processes waiting on a lock is less than 1.

Safety and liveness properties

Any property is either a *safety property*, a *liveness property*, or a conjunction of a safety and a liveness property.

Safety properties

A safety property is of the form *nothing bad happens* (that is, all states are safe).

Examples:

The number of processes in a critical section is always less than 2.

Let p be the sequence of produced values and c be the sequence of consumed values. c is always a prefix of p .

Liveness properties

A liveness property is of the form *something good happens* (that is, an interesting state is eventually achieved).

Examples:

A process that wishes to enter the critical section eventually does so.

p grows without bound.

For every value x in p , x is eventually in c .

Safety and Liveness

Showing a safety property P holds:

- find a safety property P' : $P' \Rightarrow P$;
- show that P' initially holds;
- show that each step of the program maintains P' .

Showing a liveness property holds is done by induction.

Basic properties of environment

- *Finite progress axiom*

Each process takes a step infinitely often.

Do all schedulers implement this?

If not, do such schedulers pose problems?

- *Atomic shared variables*

Consider

$\{x = A\}$
 $x = B;$
 $\{x = B\}$

any concurrent read of x will return
either A or B .

Producer/Consumer problem

Let p be the sequence of produced values and c be the sequence of consumed values.

- c is always a prefix of p .
- For every value x in p , x is eventually in c .

Bounded buffer variant:

- Always $|p| - |c| \leq \max$

Solving producer-consumer

```
int buf, produced = 0;                /* max = 1 */
```

Producer

```
while (1) {  
    while (produced) ;  
    buf = v;          /* logically appends v to p */  
    produced = 1;  
}
```

Consumer

```
while (1) {  
    while (!produced) ;  
    c = append(c, buf);  
    produced = 0;  
}
```

Mutual exclusion

- The number of processes in the critical section is never more than 1.
- If a process wishes to enter the critical section then it eventually does so.

Solving mutual exclusion

```
int in1, in2, turn = 1;

while (1) {
    in1 = 1;
    turn = 2;
    while (in2 && turn == 2) ;
    Critical section for process 1;
    in1 = 0;
}

while (1) {
    in2 = 1;
    turn = 1;
    while (in1 && turn == 1) ;
    Critical section for process 2;
    in2 = 0;
}
```

Proof of Peterson's algorithm: I

```
int in1, in2, turn = 1, at1 = 0, at2 = 0;
```

```
while (1) {  
    < in1 = 1; at1 = 1; >  
    < turn = 2; at1 = 0; >  
    while (in2 && turn == 2) ;  
    Critical section for process 1;  
    in1 = 0;  
}
```

```
while (1) {  
    < in2 = 1; at2 = 1; >  
    < turn = 1; at2 = 0; >  
    while (in1 && turn == 1) ;  
    Critical section for process 2;  
    in2 = 0;  
}
```

Proof of Peterson's algorithm: II

```
while (1) {  
    { $\neg in1 \wedge (turn = 1 \vee turn = 2) \wedge \neg at1$ }  
     $\langle in1 = 1; at1 = 1; \rangle$   
    { $in1 \wedge (turn = 1 \vee turn = 2) \wedge at1$ }  
     $\langle turn = 2; at1 = 0; \rangle$   
    { $in1 \wedge (turn = 1 \vee turn = 2) \wedge \neg at1$ }  
    while (in2 && turn == 2) ;  
    { $in1 \wedge (turn = 1 \vee turn = 2) \wedge \neg at1 \wedge (\neg in2 \vee turn = 1 \vee at2)$ }  
    Critical section for process 1;  
    { $in1 \wedge (turn = 1 \vee turn = 2) \wedge \neg at1 \wedge (\neg in2 \vee turn = 1 \vee at2)$ }  
    in1 = 0;  
    { $\neg in1 \wedge (turn = 1 \vee turn = 2) \wedge \neg at1$ }  
}
```

Proof of Peterson's algorithm: III

```
while (1) {  
    { $\neg in2 \wedge (turn = 1 \vee turn = 2) \wedge \neg at2$ }  
     $\langle in2 = 1; at2 = 1; \rangle$   
    { $in2 \wedge (turn = 1 \vee turn = 2) \wedge at2$ }  
     $\langle turn = 1; at2 = 0; \rangle$   
    { $in2 \wedge (turn = 1 \vee turn = 2) \wedge \neg at2$ }  
    while (in1 && turn == 1) ;  
    { $in2 \wedge (turn = 1 \vee turn = 2) \wedge \neg at2 \wedge (\neg in1 \vee turn = 2 \vee at1)$ }  
    Critical section for process 2;  
    { $in2 \wedge (turn = 1 \vee turn = 2) \wedge \neg at2 \wedge (\neg in1 \vee turn = 2 \vee at1)$ }  
    in2 = 0;  
    { $\neg in2 \wedge (turn = 1 \vee turn = 2) \wedge \neg at2$ }  
}
```

Proof of Peterson's algorithm: IV

at CS1 and at CS2 implies false:

$$\begin{aligned} & in1 \wedge (turn = 1 \vee turn = 2) \wedge \neg at1 \wedge (\neg in2 \vee turn = 1 \vee at2) \wedge \\ & in2 \wedge (turn = 1 \vee turn = 2) \wedge \neg at2 \wedge (\neg in1 \vee turn = 2 \vee at1) \\ & \Rightarrow (turn = 1) \wedge (turn = 2) \end{aligned}$$

process 1 in non-critical, process 2 trying to enter critical section and is blocked implies false:

$$\begin{aligned} & \neg in1 \wedge (turn = 1 \vee turn = 2) \wedge \neg at1 \wedge \\ & in2 \wedge (turn = 1 \vee turn = 2) \wedge \neg at2 \wedge \\ & in1 \wedge turn = 1 \\ & \Rightarrow \neg in1 \wedge in1 \end{aligned}$$

Proof of Peterson's algorithm: V

process 1 trying to enter critical section, process 2 trying to enter critical section, and both blocked implies false:

$$in2 \wedge turn = 2 \wedge$$

$$in1 \wedge turn = 1 \wedge$$

$$\Rightarrow (turn = 1) \wedge (turn = 2)$$

Revisiting shared variables

Both solutions use busy waiting, which is often an inefficient approach:

- A process that is busy waiting cannot proceed until some other process takes a step.
- Such a process can relinquish its processor rather than needlessly looping.
- Doing so can speed up execution.

When is this an invalid argument?

Semaphores

A *semaphore* is an abstraction that allows a process to relinquish its processor.

Two kinds: *binary* and *general*.

Binary:

P(s): `< if (s == 1) s = 0; else block; >`

V(s): `< s = 1;`
 `if (there is a blocked process) then unblock one; >`

(Edsger Dijkstra, 1965)

Semaphores: II

General:

P(s): $\langle \text{if } (s > 0) \text{ } s--; \text{ else block;} \rangle$

V(s): $\langle s++;$
 $\text{if } (\textit{there is a blocked process}) \text{ then } \textit{unblock one}; \rangle$

Solving producer-consumer

```
gensem put=max, take=0;  
int buf[max], in=0, out=0;
```

Producer

```
while (1) {  
    P(put);  
    buf[in] = v; in = (in + 1) % max;  
    V(take);  
}
```

Consumer

```
while (1) {  
    P(take);  
    c = buf[out]; out = (out + 1) % max;  
    V(put);  
}
```

Solving mutual exclusion

```
binsem cs=1;

while (1) {
    P(cs);
    Critical section for process i;
    V(cs);
}
```

Binary vs. general semaphores

gensem `s=i` is replaced with

```
struct s {  
    binsem mtx=1, blk=(i == 0) ? 0 : 1;  
    val = i;  
}
```

`P(s)` is replaced with

```
P(s.blk);  
P(s.mtx);  
if (--s.val > 0) V(s.blk);  
V(s.mtx);
```

`V(s)` is replaced with

```
P(s.mtx);  
if (++s.val == 1) V(s.blk);  
V(s.mtx);
```

Monitors

An object approach based on critical sections and explicit synchronization variables.

Entry procedures obtain mutual exclusion.

Condition variables (e.g., c):

- `c.wait()` causes process to wait outside of critical section.
- `c.signal()` causes one blocked process to take over critical section (signalling process temporarily leaves critical section).

Semaphores using monitors

```
monitor gsem {
    long value;    // value of semaphore
    condition c;  // value > 0
public:
    void entry P(void);
    void entry V(void);
    gsem(long initial);
};

gsem::gsem(long initial) { value = initial; }

void gsem::P(void) {
    if (value == 0) c.wait(); value--;
}

void gsem::V(void) {
    value++; if (value > 0) c.signal();
}
```

Solving producer-consumer

```
monitor PC {
    const long max = 10;
    long buf[max], i=0, j=0, in=0;
    condition notempty; // in > 0
    condition notfull;  // in < max
public:
    void entry put(long v);
    long entry get(void);
};

void PC::put(long v) {
    if (in == max) notfull.wait();
    buf[i] = v; i = (i + 1) % max; in++;
    notempty.signal();
}

long PC::get(void) {
    long v;
    if (in == 0) notempty.wait();
    v = buf[j]; j = (j + 1) % max; in--;
    notfull.signal();
}
```

Monitors using semaphores

Assume a monitor m with a single condition c
(easily generalized for multiple conditions).

Generate: struct m {
 binsem lock=1;
 gensem urgent=0, csem=0;
 long ccount=0, ucount=0
 };

Wrap each *entry* method:

```
P(m.lock);  
code for entry method  
if (m.ucount > 0) V(m.urgent);  
else V(m.lock);
```

Monitors using semaphores II

Replace `c.wait()` with:

```
m.ccount++;  
if (m.ucount > 0) V(m.urgent);  
else V(m.lock);  
P(m.csem);  
m.ccount--;
```

Replace `c.signal()` with:

```
m.ucount++;  
if (m.ccount > 0) {  
    V(m.csem);  
    P(m.urgent);  
}  
m.ucount--;
```

Conditions vs. semaphores

A semaphore has memory while conditions do not.

<code>V(s);</code>	<code>...</code>	
<code>...</code>	<code>P(s);</code>	<i>does not block</i>

<code>c.signal();</code>	<code>...</code>	
<code>...</code>	<code>c.wait();</code>	<i>blocks</i>

Programming with conditions

A monitor has an *invariant* (a safety property) that must hold whenever a process enters (and hence leaves) the critical region. A condition *strengthens* this invariant.

When a process requires the stronger property that doesn't hold, it waits on the condition.

When a process establishes the condition, it signals the condition.

Readers and Writers

Let r be the number of readers and w be the number of writers.

invariant I : $(0 \leq r) \wedge (0 \leq w \leq 1) \wedge (r = 0 \vee w = 0)$

readers priority version: let wr be number of waiting readers.

readOK: $I \wedge (w = 0)$

writeOK: $I \wedge (w = 0) \wedge (r = 0) \wedge (wr = 0)$

Readers and Writers II

```
monitor rprio {
    long r, w;    // number of active read/write
    long wr;      // number of waiting to read
    condition readOK;
    condition writeOK;
public:
    void entry startread(void);
    void entry endread(void);
    void entry startwrite(void);
    void entry endwrite(void);
    rprio(void);
};

rprio::rprio(void) { r = w = wr = 0; }
```

Readers and Writers III

```
rprio::startread(void) {
    if (w > 0) {wr++; readOK.wait(); wr--;}
    r++;
    readOK.signal();
}

rprio::endread(void) {
    r--; if (r == 0) writeOK.signal(); }

rprio::startwrite(void) {
    if (r > 0 || w > 0 || wr > 0) writeOK.wait();
    w++;
}

rprio::endwrite(void) {
    w--;
    if (wr > 0) readOK.signal();
    else writeOK.signal();
}
```

Final signals

A signalling process must leave the monitor when it signals.

Signals often are performed as the last statement of an entry procedure.

This is inefficient.

Could require any signals to occur only as the process leaves the entry procedure.

Does this limit the power of monitors?

Weakening wait and signal

$\{I\}$

`if ( !c ) {  $I \wedge \neg c$  } c.wait(); {  $I \wedge c$  }`

$\{I \wedge c\}$

have `c.notify()` move one process blocked on `c` to ready queue.

$\{I\}$

`while ( !c ) {  $I \wedge \neg c$  } c.wait(); {  $I$  }`

$\{I \wedge c\}$

... and, have `c.broadcast()` move *all* processes blocked on `c` to ready queue.

Weakening wait and signal, II

```
rprio::startread(void) {
    while (w > 0) {wr++; readOK.wait(); wr--; }
    r++;
}

rprio::endread(void) {
    r--; if (r == 0) writeOK.notify(); }

rprio::startwrite(void) {
    while (r > 0 || w > 0 || wr > 0)
        writeOK.wait();
    w++;
}

rprio::endwrite(void) {
    w--;
    if (wr > 0) readOK.broadcast();
    else writeOK.notify();
}
```

Moral...

Weakening an abstraction is often a good way to improve performance.

Approach used in Mesa (Xerox PARC 1970s) and in the Unix kernel.

It becomes more inefficient as the amount of contention increases, but if this holds then there are more serious problems to be addressed.