Shared Memory Programming I

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

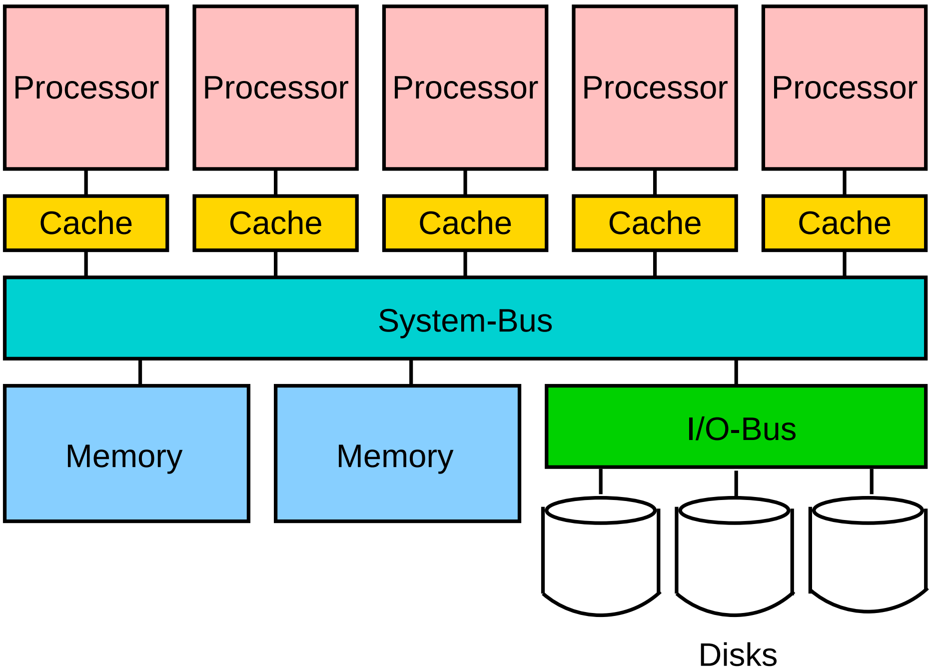
Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

Shared Memory Machines

- Multiprocessors
- Closely coupled processors
- UMA architectures

  Uniform Memory Access
- SMP architectures

  Symmetrical Multiprocessing



Example

Sequent Symmetry S81

- `melmac.risc.uni-linz.ac.at`
- 20 processors 80386/80387 @ 20MHz
- 128 MB memory
- 80 MB/s bus transfer rate.

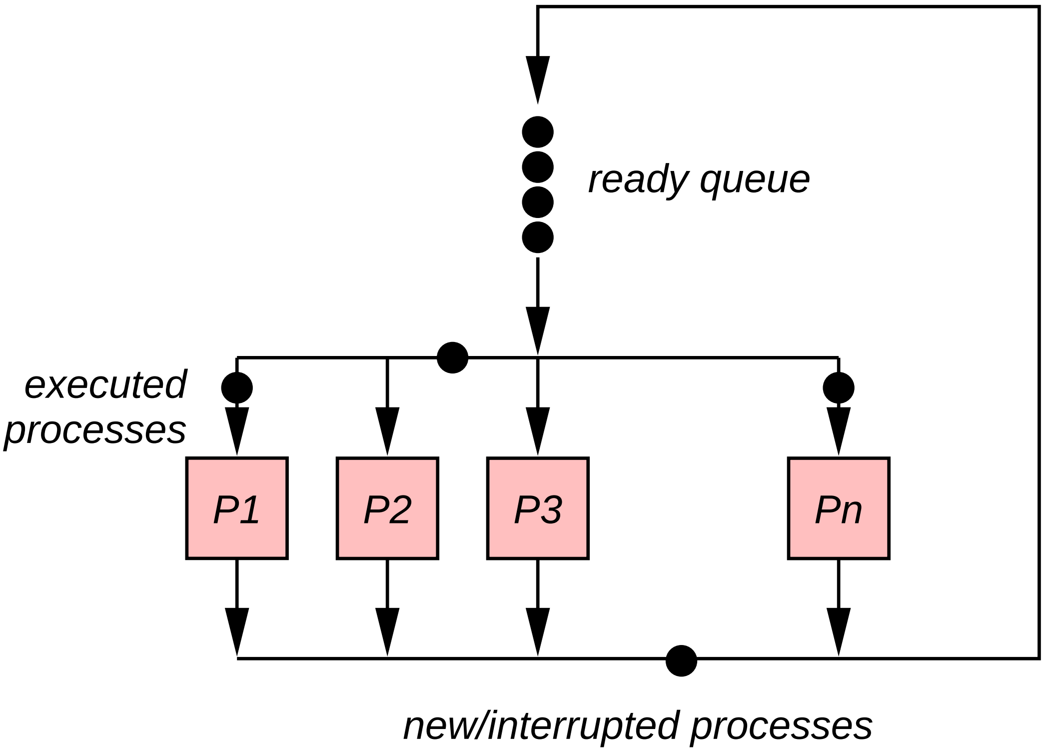
  Each processor $\approx$ 5 MIPS $*$ 1 word $\approx$ 20 MB/s maximum bus load per processor.
- 128 KB cache per processor

Symmetrical Multiprocessing

- Unix operating system
- Parallelism transparent to users

  Operated like any workstation.

- Queue of executable processes
- Automatic mapping to processors



Symmetrical Multiprocessing

Scheduling algorithm

```
foreach P[i] do in parallel
  loop forever
    get job J with higest priority
      from ready queue
    execute at most one timeslice
      of J
    if J not yet terminated then
      put J back into queue
```

Load optimally balanced among processors.

Cache Protocols

- Cache = fast buffer memory.
- Holds recently accessed data.
- Organized in lines of n memory words.

  e.g. 8192 lines $*$ 4 words $*$ 4 bytes.

Read access:

- Cache hit: word is read from cache.
- Cache miss: bus transfer takes place.
  1. Cache line read from memory.
  2. Replaces some (e.g. least recently used) line.

Write access:

- Problem: problem cache consistency.
- Processor modifies content of cache line.
- Other processors hold invalid copies.

Snoopy Bus Protocols

Cache consistency by bus watching mechanism

- Each cache watches bus for write operations
- Write through: forward write immediately to memory
  - Write-invalidate: cache copies are invalidated.
  - Write-update: cache copies are updated.
  - Disadvantage: high bus traffic.
- Copy back: write modified cache line only back if
  1. Line needs to be replaced, or
  2. Another processor reads data.

See additional slides.

Shared Memory Synchronization

Problem: need *mutual exclusion* on shared data structures

```
int stack[];
int sptr = 0;
P: loop
     v = produce()
     stack[sptr] = v  | C1
     sptr++           |
C: loop
     sptr--           | C2
     v = stack[sptr]  |
     consume(v)
```

- C_1 and C_2 are "critical regions"
- Execution must not overlap.
- Must be executed *atomically*

Synchronization Mechanisms

Lock variable ("mutex")

```
mutex lock = INIT();
P: loop                  C: loop
     v = produce()            LOCK(lock)
     LOCK(lock)               C2
     C1                       UNLOCK(lock)
     UNLOCK(lock)             consume(v)
```

- Shared variable
- States: "locked" and "unlocked"
- If locked, `LOCK()` blocks until unlocked.

Only one process can have lock ⇒ only one process can enter critical region!

Lock Variables

Implementation by "Test&Set" operation

```
typedef shared int mutex;
LOCK(mutex *lock)
  do
    old = TestAndSet(lock);
  while (old == 1)
UNLOCK(mutex *lock)
  *lock = 0
```

- *Atomic* operation `TestandSet(lock)`
  1. sets lock to 1 and
  2. returns previous value of lock.
- Implemented by hardware mechanism.

Constant polling of lock variable!

Shared Memory Synchronization

Semaphores (Dijkstra)

```
Semaphore s = INIT(1)
P1: loop      P2: loop
      P(s)          P(s)
      C1            C2
      V(s)          V(s)
```

- Counter variable holding natural number.
- Atomic operations P and V.
- $P(s)$: if $s > 0$ then $s \leftarrow s - 1$ else block.
- $V(s)$: if processes blocked on s then release some process else $s \leftarrow s + 1$.

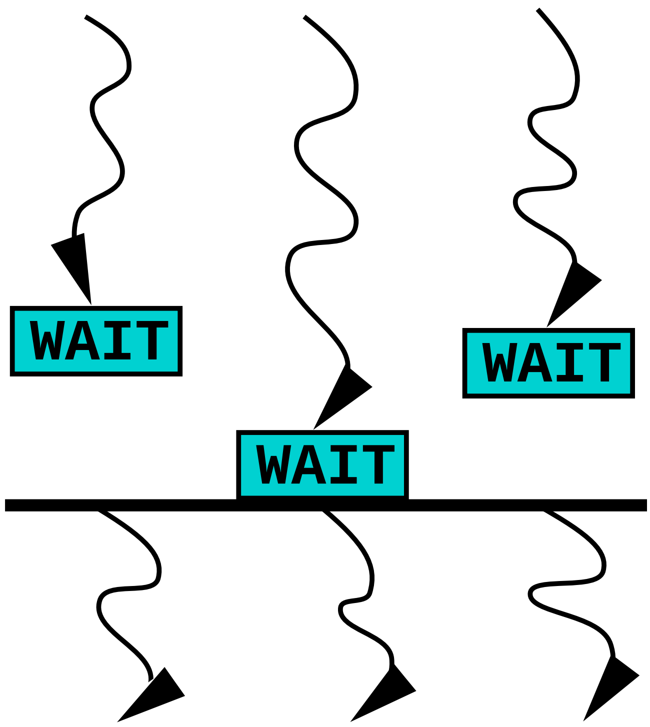
Typically implemented by locks!

Shared Memory Synchronization

Barriers

```
barrier b = INIT(n)
WAIT(barrier)
```

- Synchronization of n processes.
- `WAIT()` blocks until all n processes have reached barrier.



Shared Memory Programming

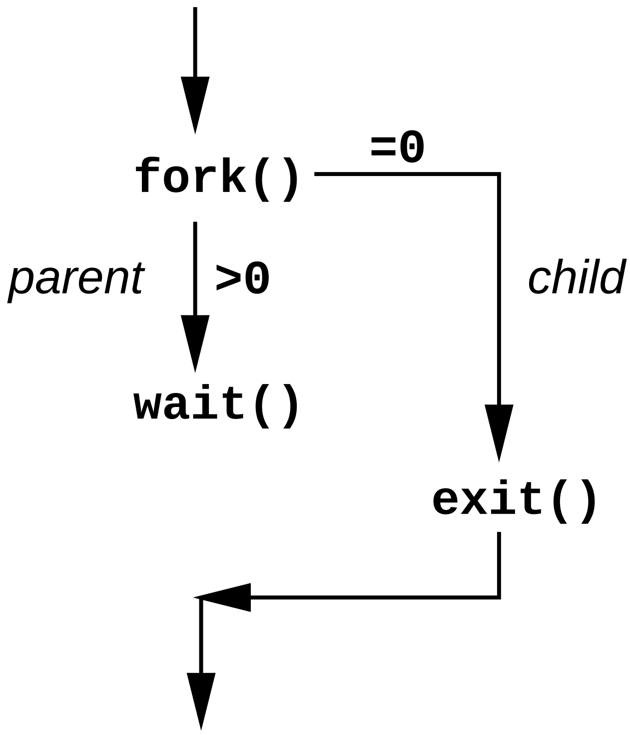
- Process creation,
- Shared memory communication,
- Synchronization.

Unix

- Process creation `fork()`
- Process synchronization `wait()`, `exit()`
- Signal handling
- System V IPC
  - Interprocess communication,
  - Messages, shared memory, semaphores.

Parallel Programming in Unix

```
pid = fork();
if (pid == 0)
  {
    ... child process
    exit(code);
  }
... parent process
wait(&status)
```



Sequent Micro-Tasking Library

Light-weight parallel programming

```
#include <parallel/parallel.h>
#include <parallel/microtask.h>

cc -O prog.c -o prog -lpps -lseq
```

- Shared memory allocation

```
shared float a[N][N];
a = shmalloc(N*N*sizeof(float));
```

- Process creation

```
m_set_procs(nproc);
m_fork(fun, a1, ..., an);
m_kill_procs();
```

All micro-tasks execute (asynchronously) the same program $\Rightarrow$ Single Program Multiple Data (SPMD) model.

Sequent Micro-Tasking Library

- Process information

```
nproc = m_get_numprocs();
myid = m_get_myid();
```

$0 \leq \text{myid} < \text{nproc}$

- Locks

```
m_lock();
m_unlock();
```

Default lock used.

- Barriers

```
m_sync();
```

Default barrier used.

Sequent Parallel Programming

- Timing: `getusclk()`

  See manual pages

- Process handling

```
/* show all my processes */
ps -fu <my_login>

/* start process in background */
command &

/* terminate process */
kill -9 <pid>
```

- Machine monitoring

```
/etc/monitor
```